Institute for Plasma Research

प्लाज़्मा अनुसंधान संस्थान
Institute for Plasma Research

REPORT

ARM CONTROLLER BASED DATA LOGGING SYSYTEM

Prepared by: Misha Joshi
Jeet Bhatt

Guided by: Vismaysinh Raulji
Praveenlal E V

ARM Controller Based Data Logging System

Abstract

This project is meant for studying the ARM Cortex M3 architecture of high end microcontroller. After getting acquainted with the architecture, the exercise of writing the C program and testing the application of this MCU on STM32Cube IDE will be performed, and further exercise eventually continue to explore all important peripherals of MCU like GPIO, UART, SPI and ADC. Tasks will be performed using higher level hardware abstraction layer (HAL) APIs to access the hardware peripherals and it is expected that the GUI/HMI will be developed in LabVIEW/Python to interface mcu over UART to computer.

The required interfaces with the different peripherals will be tested. The final goal of the project is to display the ADC input data on GUI/HMI. The complete chain of data transfer involving ADC, UART, LabVIEW/Python to be developed and tested for desired functionality.

Acknowledgement

The opportunity of working on an academic project at IPR was a great chance for expanding our knowledge and gave us exposure to the professional environment. We would like to extend our deepest gratitude to our mentor **Praveenlal E V Sir** and project guide **Vismaysinh Raulji Sir**, for their guidance and insightful suggestions throughout the project. Their expertise and encouragement have been instrumental in shaping the direction of this project and refining its outcomes.

We would also like to extend our gratitude to **Mitesh Patel Sir**, **Pramila Gautam Ma'am**, **Praveena Shukla Ma'am** and finally to **Rachana Rajpal Ma'am** of Institute for Plasma Research who have provided access to resources and technical assistance as and when needed.

Introduction

The ARM Controller Based Data Logging System project aims to develop an efficient prototype for capturing and storing data in various applications. This project utilizes an ARM (Advanced RISC Machine) microcontroller to provide reliable and accurate data logging capabilities for a wide range of industries and domains.
Data logging systems have various applications across multiple industries suc as Environmental Monitoring, Energy Management, Industrial Process Control, Research and Development, Vehicle Diagnostics, Agriculture and Farming, Asset Tracking, Home Automation, etc. These are just a few examples of the numerous applications of data logging systems. The versatility and flexibility of these systems make them invaluable across industries where accurate and reliable data collection is necessary for analysis, optimization, and decision-making.
We started this academic project by programming a Nucleo F070RB development board having an ARM Cortex M0 architecture, using STM32Cube IDE. After achieving this, we developed a GUI in LabVIEW for the same.

INDEX

1. NUCLEO F070RB **1**

1.1. Microcontroller features 1

1.2. Nucleo Features 1

1.3. Board Pinout 2

1.4. UART 4

1.5. Timers 5

1.6. Interrupts 6

1.7. ADC 7

2. STM32Cube IDE **9**

2.1. Introduction 9

2.2. HAL Library 9

2.3. Getting acquainted with the user interface 10

3. LabVIEW **29**

3.1. Introduction 29

3.2. Advantages of LabVIEW 29

3.3. Features of LabVIEW 29

3.4. Graphical user interface 30

4. Applications **34**

5. Conclusion **36**

6. References **37**

1. NUCLEO-F070RB

1.1. Microcontroller Features

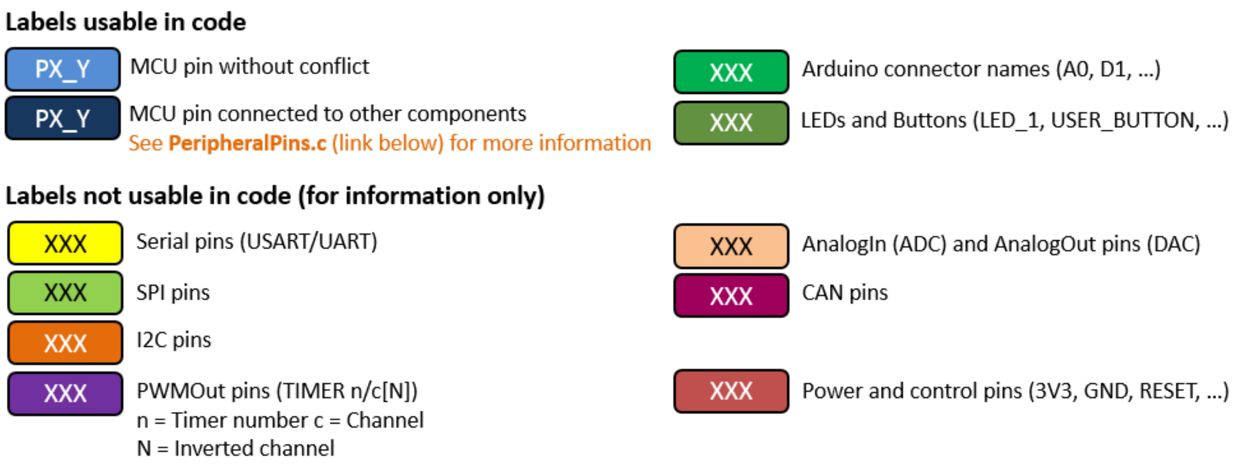
- STM32F070RBT6 in LQFP64 package
- ARM®32-bit Cortex®-M0 CPU
- 48 MHz max CPU frequency
- VDD from 2.4 V to 3.6 V
- 128 KB Flash
- 16 KB SRAM
- GPIO (51) with external interrupt capability
- 12-bit ADC with 16 channels
- RTC
- Timers (8)
- I2C (2)
- USART (4)
- SPI (2)
- USB 2.0 full-speed

1.2. Nucleo features

- Two types of extension resources
  - Arduino Uno Revision 3 connectivity
  - STMicroelectronics Morpho extension pin headers for full access to all STM32 I/Os
- On-board ST-LINK/V2-1 debugger/programmer with SWD connector
  - Selection-mode switch to use the kit as a standalone ST-LINK/V2-1
- Flexible board power supply
- USB VBUS or external source (3.3 V, 5 V, 7 - 12 V)
  - Power management access point
- User LED (LD2)
- Two push buttons: USER and RESET
- USB re-enumeration capability: three different interfaces supported on USB
  - Virtual Com port
  - Mass storage (USB Disk drive) for drag and drop programming
  - Debug port

1.3. Board Pinout

Pins Legend



Labels usable in code

PX_Y	MCU pin without conflict
PX_Y	MCU pin connected to other components See **PeripheralPins.c** (link below) for more information
XXX	Arduino connector names (A0, D1, ...)
XXX	LEDs and Buttons (LED_1, USER_BUTTON, ...)

Labels not usable in code (for information only)

XXX	Serial pins (USART/UART)
XXX	SPI pins
XXX	I2C pins
XXX	PWMOut pins (TIMER n/c[N]) n = Timer number c = Channel N = Inverted channel
XXX	AnalogIn (ADC) and AnalogOut pins (DAC)
XXX	CAN pins
XXX	Power and control pins (3V3, GND, RESET, ...)

Fig. 1.1

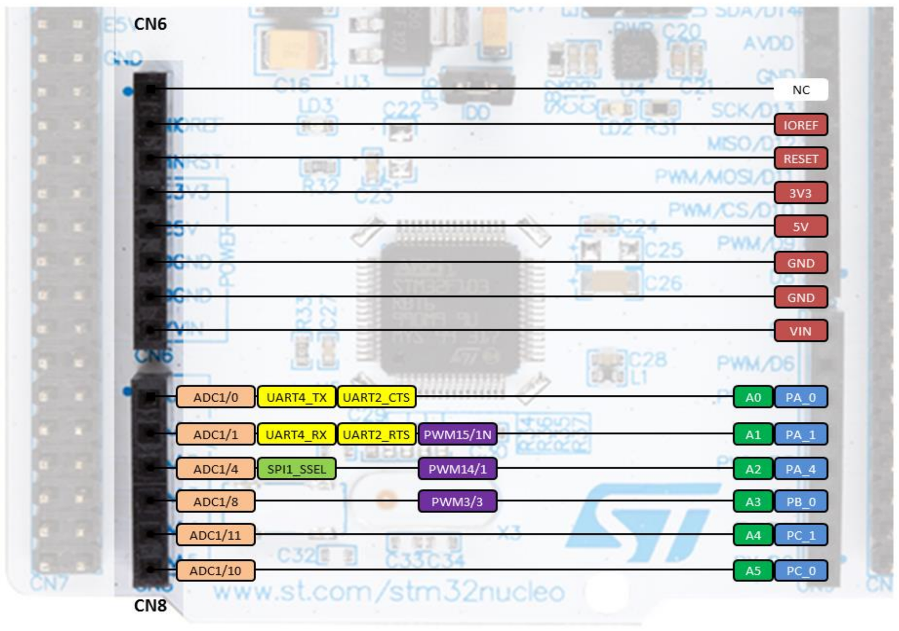


Fig. 1.2

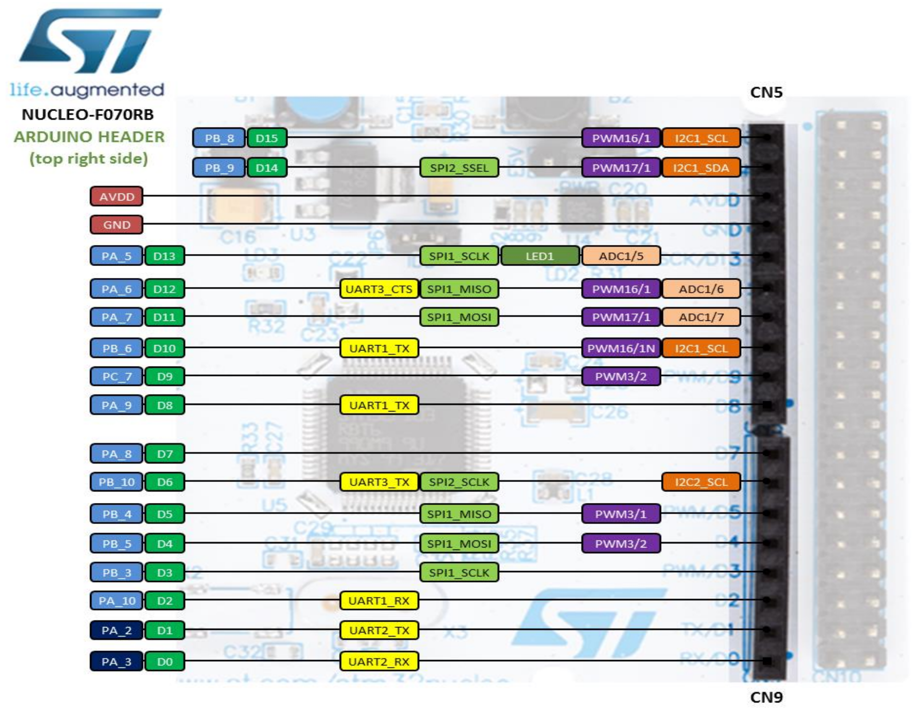

life.augmented
NUCLEO-F070RB
ARDUINO HEADER
(top right side)
CN5
PB_8 D15 PWM16/1 I2C1_SCL
PB_9 D14 SPI2_SSEL PWM17/1 I2C1_SDA
AVDD
GND
PA_5 D13 SPI1_SCLK LED1 ADC1/5
PA_6 D12 UART3_CTS SPI1_MISO PWM16/1 ADC1/6
PA_7 D11 SPI1_MOSI PWM17/1 ADC1/7
PB_6 D10 UART1_TX PWM16/1N I2C1_SCL
PC_7 D9 PWM3/2
PA_9 D8 UART1_TX
PA_8 D7
PB_10 D6 UART3_TX SPI2_SCLK I2C2_SCL
PB_4 D5 SPI1_MISO PWM3/1
PB_5 D4 SPI1_MOSI PWM3/2
PB_3 D3 SPI1_SCLK
PA_10 D2 UART1_RX
PA_2 D1 UART2_TX
PA_3 D0 UART2_RX
CN9

Fig. 1.3

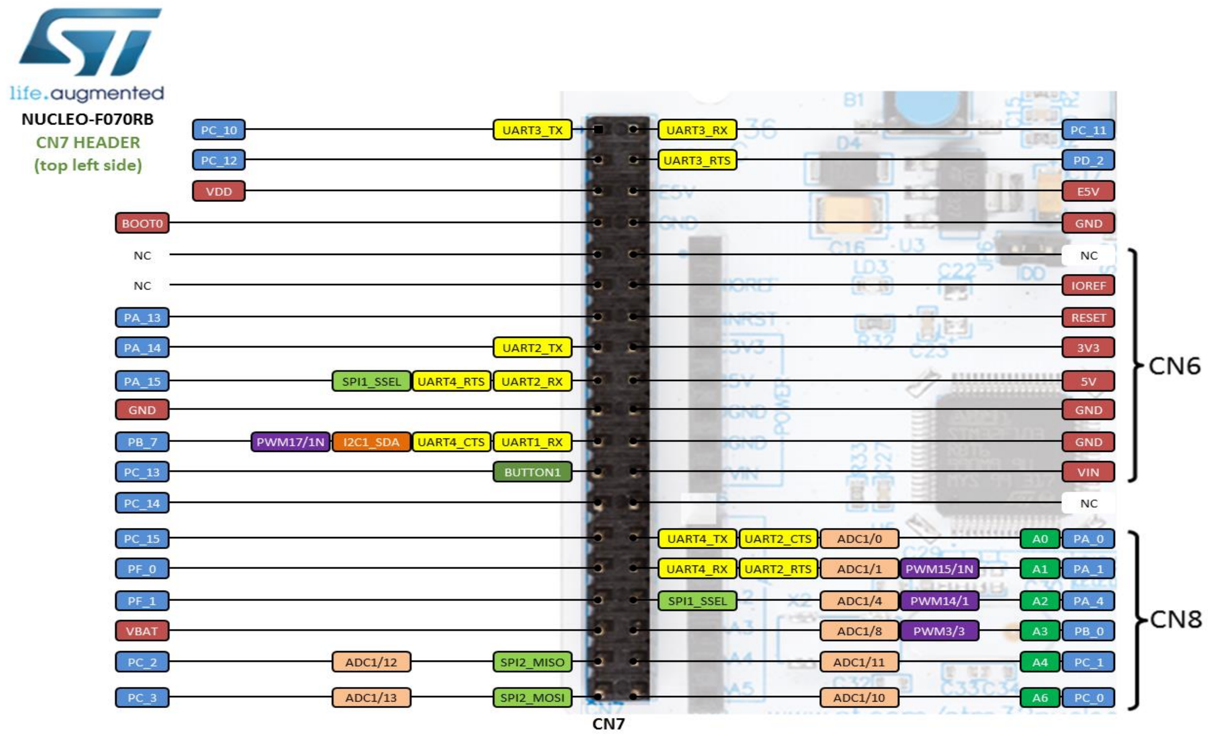

life.augmented
NUCLEO-F070RB
CN7 HEADER
(top left side)
PC_10 UART3_TX UART3_RX PC_11
PC_12 UART3_RTS PD_2
VDD E5V
BOOT0 GND
NC NC
NC IOREF
PA_13 RESET
PA_14 UART2_TX 3V3
PA_15 SPI1_SSEL UART4_RTS UART2_RX 5V
GND GND
PB_7 PWM17/1N I2C1_SDA UART4_CTS UART1_RX GND
PC_13 BUTTON1 VIN
PC_14 NC
PC_15 UART4_TX UART2_CTS ADC1/0 A0 PA_0
PF_0 UART4_RX UART2_RTS ADC1/1 PWM15/1N A1 PA_1
PF_1 SPI1_SSEL ADC1/4 PWM14/1 A2 PA_4
VBAT ADC1/8 PWM3/3 A3 PB_0
PC_2 ADC1/12 SPI2_MISO ADC1/11 A4 PC_1
PC_3 ADC1/13 SPI2_MOSI ADC1/10 A6 PC_0
CN6
CN8
CN7

Fig. 1.4

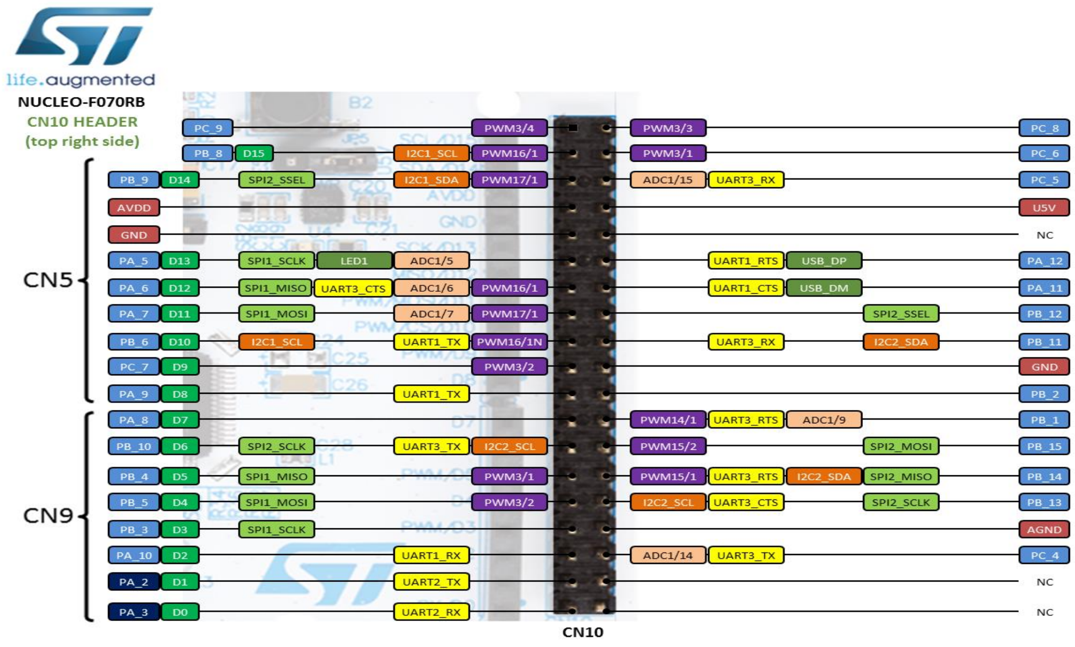

life.augmented
NUCLEO-F070RB
CN10 HEADER
(top right side)
CN5
CN9
CN10
PC_9 PWM3/4 PWM3/3 PC_8
PB_8 D15 I2C1_SCL PWM16/1 PWM3/1 PC_6
PB_9 D14 SPI2_SSEL I2C1_SDA PWM17/1 ADC1/15 UART3_RX PC_5
AVDD U5V
GND NC
PA_5 D13 SPI1_SCLK LED1 ADC1/5 UART1_RTS USB_DP PA_12
PA_6 D12 SPI1_MISO UART3_CTS ADC1/6 PWM16/1 UART1_CTS USB_DM PA_11
PA_7 D11 SPI1_MOSI ADC1/7 PWM17/1 SPI2_SSEL PB_12
PB_6 D10 I2C1_SCL UART1_TX PWM16/1N UART3_RX I2C2_SDA PB_11
PC_7 D9 PWM3/2 GND
PA_9 D8 UART1_TX PB_2
PA_8 D7 PWM14/1 UART3_RTS ADC1/9 PB_1
PB_10 D6 SPI2_SCLK UART3_TX I2C2_SCL PWM15/2 SPI2_MOSI PB_15
PB_4 D5 SPI1_MISO PWM3/1 PWM15/1 UART3_RTS I2C2_SDA SPI2_MISO PB_14
PB_5 D4 SPI1_MOSI PWM3/2 I2C2_SCL UART3_CTS SPI2_SCLK PB_13
PB_3 D3 SPI1_SCLK AGND
PA_10 D2 UART1_RX ADC1/14 UART3_TX PC_4
PA_2 D1 UART2_TX NC
PA_3 D0 UART2_RX NC

Fig. 1.5

1.4. UART

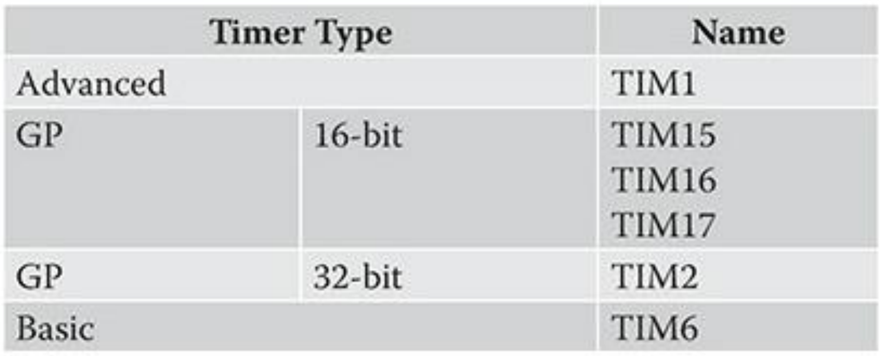
MCU uses bit serial communication to bidirectionally transfer data between it and external devices and systems. The primary peripheral devices used for this function are the universal asynchronous receiver transmitter (UART) and the universal synchronous/asynchronous receiver/transmitter (USART). The principal difference between these peripherals is that the USART can use a synchronizing clock pulse train between nodes, while the UART is completely self-synchronizing. Full-duplex communications means that data can simultaneously be sent and received between nodes. This is strictly possible because each node’s UART or USART is controlled by its own MCU. The UART peripherals in the microcontroller can be used to send serial data to the PC serial COM port and display it on a terminal.

1.5. TIMERS

Timer peripherals are very important components within a MCU. Many embedded applications are time or temporal dependent and timers are the primary means by which the MCU controls the application. While it is certainly possible to use a MCU to directly time processes, it would be a great waste of processing power and highly inefficient approach. Using hardware timers along with interrupts is really the only practical way to implement embedded time-dependent applications.

Timers have many uses, including but not limited to the following:
• Generating a precise time base. All STM timers can do this.
• Measuring the frequency of an incoming digital pulse train.
• Measuring elapsed time on an output signal. This is called output compare for STM timers.
• Generating precise pulse-width modulation (PWM) signals used for servo and motor control.
• Generating single pulse with programmable length and delay characteristics.
• Generating periodic direct memory access (DMA) signals in response to update, trigger, input capture, and output compare events.

There are following five broad categories of STM timers:
• Basic: Simple 16-bit timers. They do not have inputs or output pins. They are principally used as "masters" for other timers. They also are used as a digital-to-analog converter (DAC) clock source. They can generate a time base like all STM timers can do.
• General purpose (GP): These are 16-bit or 32-bit timers with both input and output pins. They can do all of the functions described in the above list. GP timers can have up to four programmable channels.
• Advanced: All the features of the GP timer with additional functions related to motor control and digital power conversion. There are three complementary outputs available with this timer category with an emergency shutdown input.



Timer Type		Name
Advanced		TIM1
GP	16-bit	TIM15 TIM16 TIM17
GP	32-bit	TIM2
Basic		TIM6

Fig. 1.6 Types of timers

The output time interval is determined by the Prescaler value, the clock frequency, and the timer preload register's value.

T_{OUT} = Prescaler * Preload / F_{CLK}

1.6. INTERRUPTS

An interrupt from a high-level perspective is any process that momentarily stops an ongoing program execution and requires the MCU to execute specialized code related to the event that initiated the interrupt. This specialized code is known as an interrupt handler. The event that causes the interrupt is termed an asynchronous event, meaning that it has no causal relationship with the normally executing program code.

An interrupt can originate with hardware, which is the case when a timer is involved. It can also be purely software driven, which will often happen when an abnormal program termination is detected, such as an attempt to divide by zero or attempting to access a nonexistent memory location. Software interrupts in most computer languages are normally referred to as exceptions. However, in ARM terminology, all interrupts whether they are hardware or software initiated are called exceptions.

The ARM Cortex-M integrated peripheral (IP) contains a hardware device called the nested vector interrupt controller (NVIC) whose purpose is to manage exceptions.

A single line labeled NMI stands for non-maskable interrupt and is used to allow a peripheral to signal the NVIC to unconditionally start the interrupt process. The other interrupt type is IRQ, which stands for interrupt request. This interrupt type can be masked, meaning that the NVIC does not have to immediately respond to an

IRQ if a corresponding masking IRQ register bit has been set. There is usually only one NMI, but there are many IRQ lines that are dependent on the specific model NVIC capacity.

The NVIC is fully integrated into the HAL software framework and uses its own C structures to set up a variety of registers.The STM NVIC general specification states that up to 256 interrupt channels can be managed by a single NVIC. Out of the 256 channels, 240 may be external, while 16 are always internal.

External interrupts such as those which are sourced through GPIO pins are connected to the NVIC using an external interrupt controller (EXTI). Using an EXTI allows for multiplexing of many GPIO pins into the NVIC external interrupt inputs.

1.7. ADC

An ADC (Analog-To-Digital) converter is an electronic circuit that takes in an analog voltage as input and converts it into digital data, a value that represents the voltage level in binary code. The ADC samples the analog input whenever you trigger it to start conversion. And it performs a process called quantization so as to decide on the voltage level and its binary code that gets pushed in the output register.
The ADC does the counter operation that of a DAC, while an ADC (A/D) converts analog voltage to digital data the DAC (D/A) converts digital numbers to the analog voltage on the output pin.
The STM32 has a 12-bit ADC which is a successive approximation analog-to-digital converter.

The analog input voltage (VIN) is held on a track/hold. To implement the binary search algorithm, the N-bit register is first set to midscale (that is, 100... .00, where the MSB is set to 1). This forces the DAC output (VDAC) to be VREF/2, where VREF is the reference voltage provided to the ADC. A comparison is then performed to determine if VIN is less than, or greater than, VDAC. If VIN is greater than VDAC, the comparator output is a logic high, or 1, and the MSB of the N-bit register remains at 1. Conversely, if VIN is less than VDAC, the comparator output is a logic low and the MSB of the register is cleared to logic 0. The SAR control logic then moves to the next bit down, forces that bit high, and does another comparison. The sequence continues all the way down to the LSB. Once this is done, the conversion is complete and the N-bit digital word is available in the register.

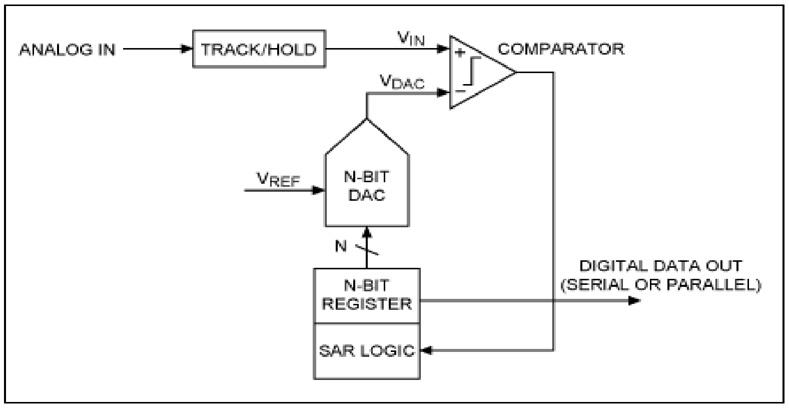


Fig. 1.7 SAR

ADC Features

- 12-bit resolution
- Interrupt generation at End of Conversion, End of Injected conversion and Analog watchdog event
- Single and continuous conversion modes
- Scan mode for automatic conversion of channel 0 to channel ‘n’
- Self-calibration
- Data alignment with in-built data coherency
- Channel by channel programmable sampling time
- External trigger option for both regular and injected conversion
- Discontinuous mode
- Dual-mode (on devices with 2 ADCs or more)
- ADC conversion time: 1 µs at 56 MHz (1.17 µs at 72 MHz)
- ADC supply requirement: 2.4 V to 3.6 V
- ADC input range: VREF– ≤ VIN ≤ VREF+
- DMA request generation during regular channel conversion

2. STM32Cube IDE

2.1. Introduction

- STM32CubeIDE is an advanced C/C++ development platform with peripheral configuration, code generation, code compilation, and debug features for STM32 microcontrollers and microprocessors. It is based on the Eclipse®/CDT™ framework and GCC toolchain for the development, and GDB for the debugging. It allows the integration of the hundreds of existing plugins that complete the features of the Eclipse® IDE.
- STM32CubeIDE integrates STM32 configuration and project creation functionalities from STM32CubeMX to offer all-in-one tool experience and save installation and development time. After the selection of an empty STM32 MCU or MPU, or pre configured microcontroller or microprocessor from the selection of a board or the selection of an example, the project is created and initialization code generated. At any time during the development, the user can return to the initialization and configuration of the peripherals or middleware and regenerate the initialization code with no impact on the user code.
- STM32CubeIDE also includes standard and advanced debugging features including views of CPU core registers, memories, and peripheral registers, as well as live variable watch, Serial Wire Viewer interface, or fault analyzer.

2.2. HAL Library

STM32Cube IDE provides the developer with all the low-level drivers, APIs. The STM32Cube Hardware Abstraction Layer (HAL), an STM32 abstraction layer embedded software ensuring maximized portability across the STM32 microcontroller. The HAL is available for all the hardware peripherals. The HAL offers high-level and feature-oriented APIs, with a high-portability level. They hide the MCU and peripheral complexity to the end-user.
The HAL driver layer provides a simple, generic multi-instance set of APIs (application programming interfaces) to interact with the upper layer (application, libraries and stacks).
The HAL drivers include a complete set of ready-to-use APIs that simplify the user application implementation. For example, the communication peripherals contain APIs to initialize and configure the peripheral, manage data transfers in polling mode, handle interrupts or DMA, and manage communication errors.

The HAL drivers are designed to offer a rich set of APIs and to interact easily with the application upper layers. Each driver consists of a set of functions covering the most common peripheral features. The development of each driver is driven by a common API which standardizes the driver structure, the functions and the parameter names. The HAL drivers include a set of driver modules, each module being linked to a standalone peripheral. However, in some cases, the module is linked to a peripheral functional mode. As an example, several modules exist for the USART peripheral like the UART driver module and USART driver module.

2.3. Getting acquainted with the User Interface

Program statement

Generate a program that meets the following conditions:

1. to always display a random character or text
2. on receiving a pre decided character, display the scaled ADC output
3. On receiving another pre decided character, stop displaying the output
4. take user input to vary the sampling intervals
5. For a given user input, vary the sampling rate using timer/adc

Solution

Configuration:

In the .ioc file, select PA0 as ADC_IN0 which is ADC input with channel 0. From timers, select TIM16 and set the prescaler and counter period according to the clock value. From connectivity, select USART2, set the mode to Asynchronous, set a baud rate and select USART2 Global Interrupt in NVIC Settings.

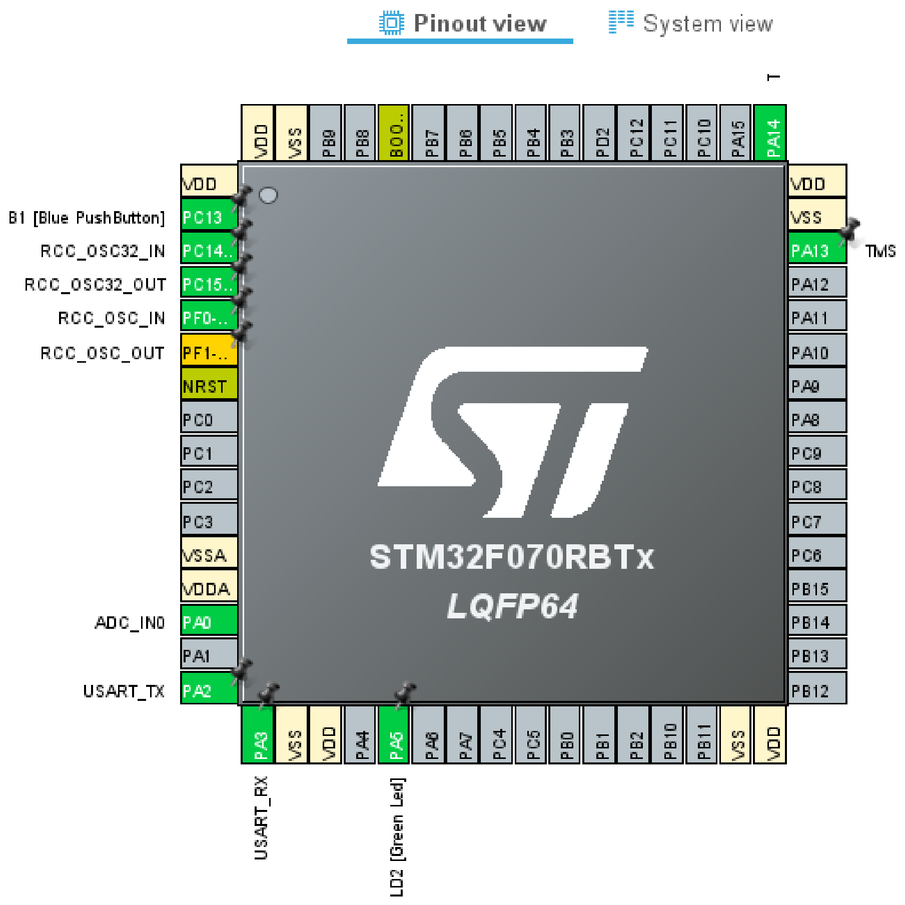
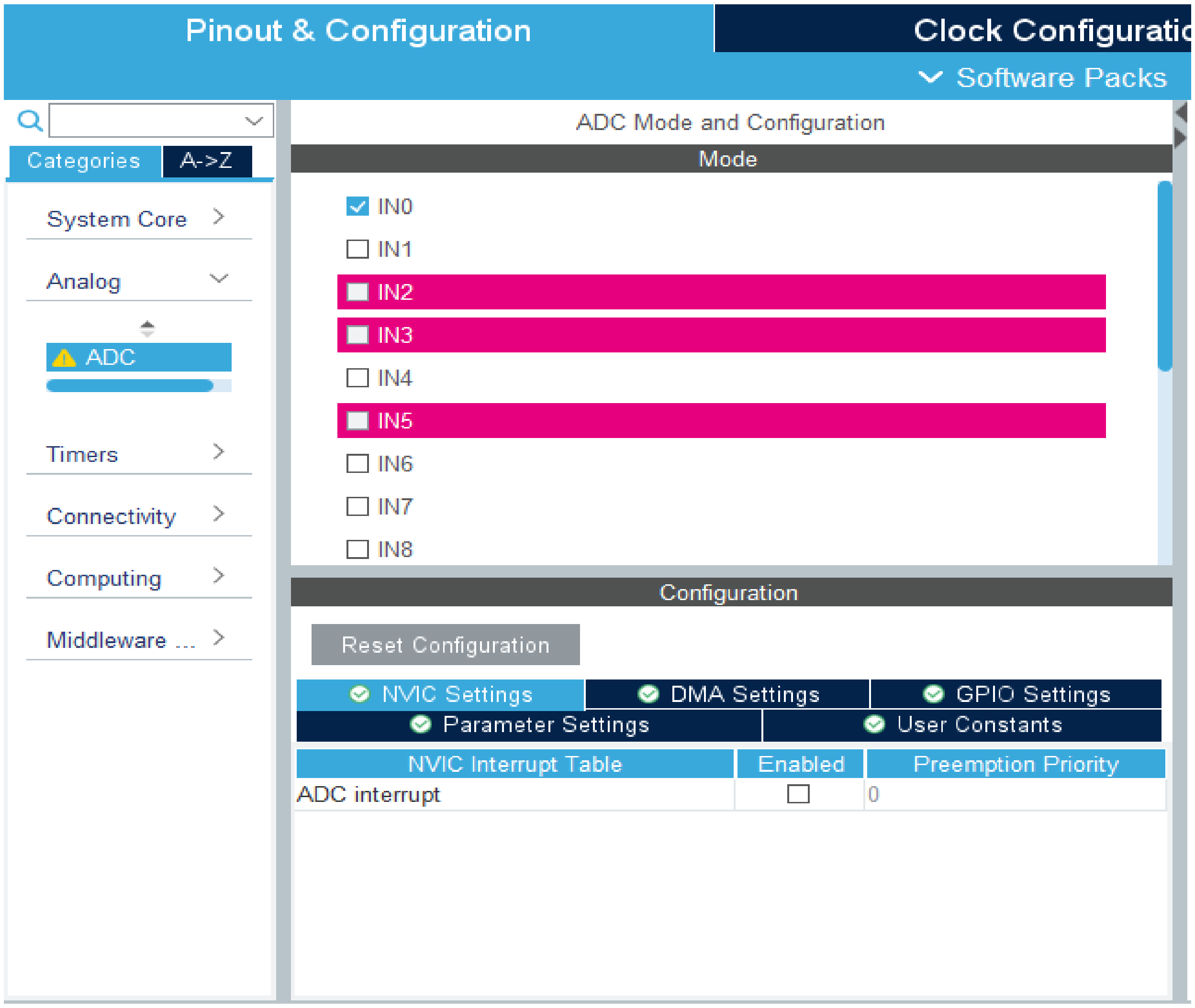


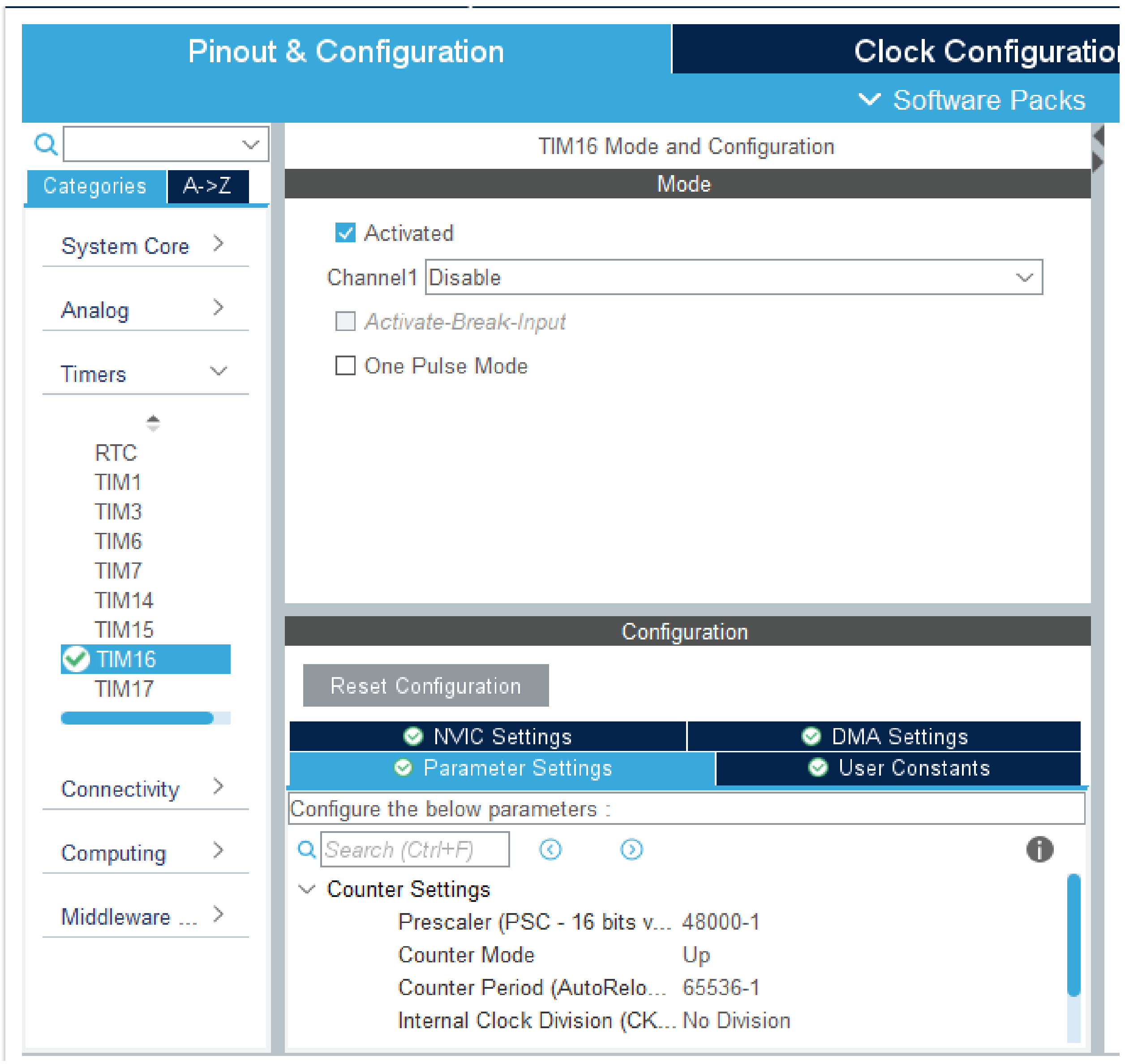
Fig. 2.1 Pinout View



- [✓] IN0
- [ ] IN1
- [ ] IN2
- [ ] IN3
- [ ] IN4
- [ ] IN5
- [ ] IN6
- [ ] IN7
- [ ] IN8

NVIC Interrupt Table	Enabled	Preemption Priority
ADC interrupt	☐	0

Fig. 2.2 ADC Configuration



TIM16 Mode and Configuration

[✓] Activated

Channel1 Disable

[ ] Activate-Break-Input

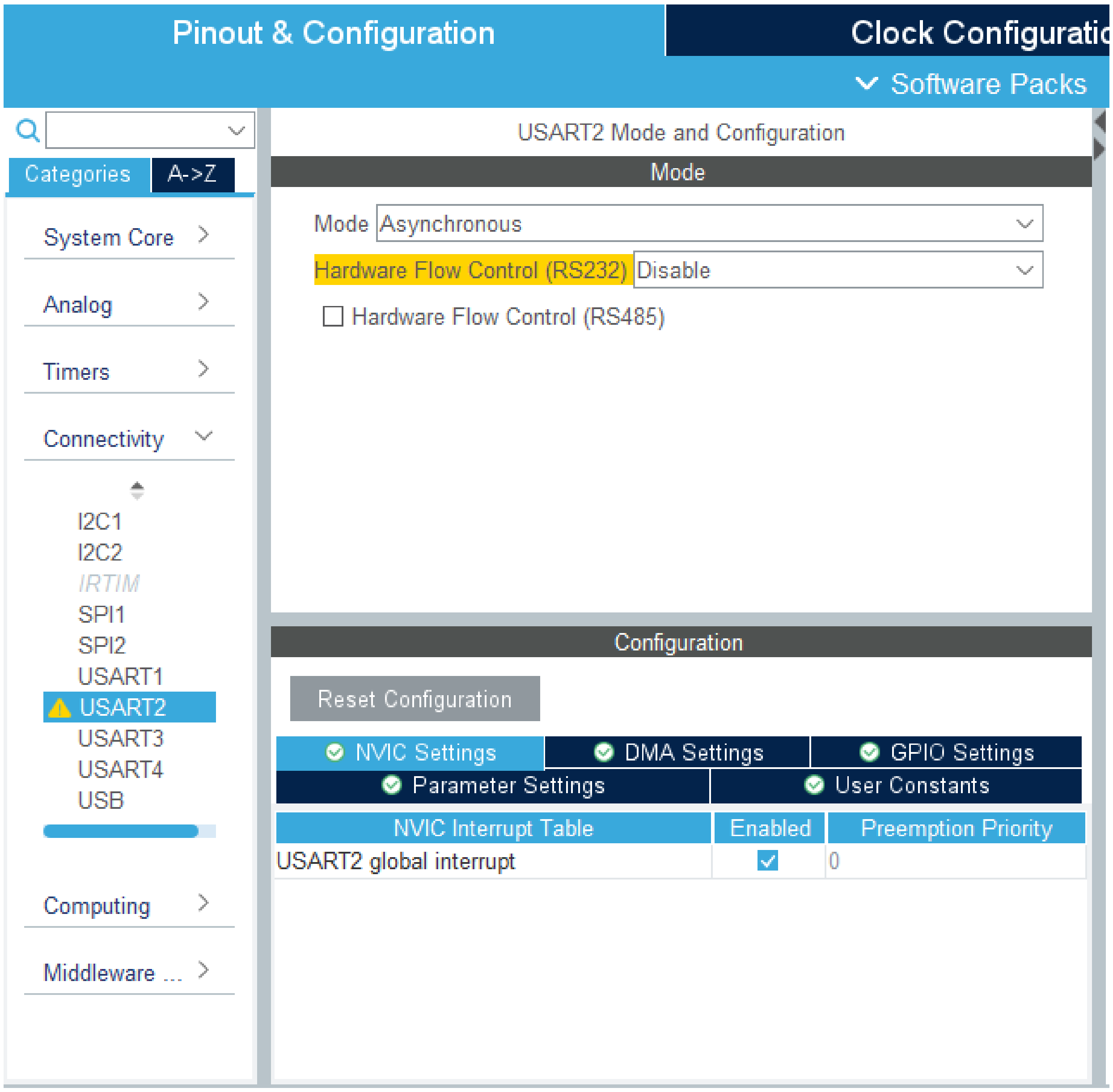
[ ] One Pulse Mode

Configure the below parameters :

Counter Settings

```
Prescaler (PSC - 16 bits v...   48000-1
Counter Mode                    Up
Counter Period (AutoRelo...     65536-1
Internal Clock Division (CK...  No Division
```

Fig. 2.3 Timer Configuration



USART2 Mode and Configuration

Mode

Mode: Asynchronous

Hardware Flow Control (RS232): Disable

☐ Hardware Flow Control (RS485)

Configuration

NVIC Interrupt Table	Enabled	Preemption Priority
USART2 global interrupt	✓	0

Fig. 2.4 USART Configuration

Program:

```
/* USER CODE BEGIN Header */
/**

*************************************************************************
******
  * @file           : main.c
  * @brief          : Main program body

*************************************************************************
******
  * @attention
  *
  * Copyright (c) 2023 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *

*************************************************************************
******
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc;

TIM_HandleTypeDef htim16;

UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_ADC_Init(void);
static void MX_TIM16_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
uint16_t timer_val;
uint8_t rx_buffer[6]; //received data will be stored in this variable
double value = 0.0;
double clr = 0.0;
uint8_t msg[10] = {'\0'};
uint8_t tx_data[67] = "Enter the sampling rate: \n\r a:10ms\n\r b:100ms\n\r c:500ms\n\r
d:1000ms\n\r"; //data to be transmitted

/* USER CODE END 0 */
```

```
/**
  * @brief  The application entry point.
  * @retval int
  */
void adc_output(void) //user defined function to transmit the value of adc output
{
               sprintf(msg, "%f\r\n", value); //the data stored in value is copied into msg
             HAL_UART_Transmit(&huart2, msg ,10 , 2); //data stored in message is
transmitted
             timer_val = __HAL_TIM_GET_COUNTER(&htim16); //current timer value is
stored in the variable
}

void end_logging(void) //user defined function to stop the data logging
{
            rx_buffer[1]=0;
            sprintf(msg, "%f\r\n", clr);
            HAL_UART_Transmit(&huart2, msg ,10 , 2);
            rx_buffer[0]=0; //rx_buffer is cleared
            rx_buffer[2]=0;
}
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();
  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_ADC_Init();
  MX_TIM16_Init();
  /* USER CODE BEGIN 2 */
  HAL_TIM_Base_Start(&htim16);
 timer_val = __HAL_TIM_GET_COUNTER(&htim16); //storing the current count of
timer in a variable

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */

  while (1){
                        HAL_ADC_Start(&hadc); //start the adc
                        HAL_ADC_PollForConversion(&hadc,100);
                        value = (double) HAL_ADC_GetValue(&hadc); //store the converted
value in a variable
                        value = value*(3.3/4096);
                     HAL_UART_Receive_IT(&huart2, rx_buffer, sizeof(rx_buffer)); //
nucleo board receives data from the terminal


                            if(rx_buffer[0]=='R') //checks if R is received
                            {

                                    if (rx_buffer[1]=='a') //checks if a is received
                                    {

                                            while(__HAL_TIM_GET_COUNTER(&htim16)-
                                            timer_val >=10) // timer value = 10ms
                                              {
```

```
adc_output(); //user defined function to transmit the value is called
                                                   }
                                                   if(rx_buffer[2]=='S') //if S is received to Stop
                                                   {
                                                          end_logging(); //user defined function to end
the data logging is called
                                                    }

                                          }

//if b is received then data is logged every 100ms and on sending 'S', data logging will end
                                         else if (rx_buffer[1]=='b')
                                         {
                                                   while(__HAL_TIM_GET_COUNTER(&htim16)-
timer_val >=100)
                                                   {
                                                          adc_output();
                                                   }
                                                   if(rx_buffer[2]=='S')
                                                  {
                                                          end_logging();
                                                   }
                                         }

//if c is received then data is logged every 500ms and on sending 'S', data logging will end
                                         else if (rx_buffer[1]=='c')
                                         {
                                                 while (__HAL_TIM_GET_COUNTER(&htim16) -
timer_val >=500)
                                                   {
                                                          adc_output();
                                                   }
                                                   if(rx_buffer[2]=='S')
                                                   {
                                                          end_logging();
                                                   }
                                         }

//if d is received then data is logged every 1000ms and on sending 'S', data logging will end
```

```
                                        else if (rx_buffer[1]=='d')
                                        {
                                                while  (__HAL_TIM_GET_COUNTER(&htim16) -
timer_val >=1000)
                                                 {
                                                        adc_output();
                                                 }
                                                 if(rx_buffer[2]=='S')
                                                 {
                                                        end_logging();
                                                 }
                                         }

//this is a default case when no character is received
                                        else
                                        {
                                                while (__HAL_TIM_GET_COUNTER(&htim16) -
        timer_val >=1000)
                                                {
                                                        adc_output();
                                                }
                                                if(rx_buffer[2]=='S')
                                                {
                                                        end_logging();
                                                }
                                         }
                                }

//The data stored in tx_data will be transmitted until 'R' is received or when rx_buffer[0]
!= 'R'
                                else
                                {
                                        HAL_UART_Transmit_IT(&huart2, tx_data,67);
                                        HAL_Delay(1000); //soft delay of 1000ms
                                 }


                }
}
/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */

RCC_OscInitStruct.OscillatorType=RCC_OSCILLATORTYPE_HSI|RCC_OSCILLAT
ORTYPE_HSI14;
RCC_OscInitStruct.HSIState = RCC_HSI_ON;
RCC_OscInitStruct.HSI14State = RCC_HSI14_ON;
RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
RCC_OscInitStruct.HSI14CalibrationValue = 16;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL6;
RCC_OscInitStruct.PLL.PREDIV = RCC_PREDIV_DIV1;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
   }

  /** Initializes the CPU, AHB and APB buses clocks
  */
RCC_ClkInitStruct.ClockType=RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYS
CLK|RCC_CLOCKTYPE_PCLK1;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;




if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
  {
    Error_Handler();
    }
  }

/**
  * @brief ADC Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC_Init(void)
{

  /* USER CODE BEGIN ADC_Init 0 */

  /* USER CODE END ADC_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC_Init 1 */

  /* USER CODE END ADC_Init 1 */

  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and
number of conversion)
  */
  hadc.Instance = ADC1;
  hadc.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
  hadc.Init.Resolution = ADC_RESOLUTION_12B;
  hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD;
  hadc.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  hadc.Init.LowPowerAutoWait = DISABLE;
  hadc.Init.LowPowerAutoPowerOff = DISABLE;
  hadc.Init.ContinuousConvMode = DISABLE;
  hadc.Init.DiscontinuousConvMode = DISABLE;
  hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START;

hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc.Init.DMAContinuousRequests = DISABLE;
  hadc.Init.Overrun = ADC_OVR_DATA_PRESERVED;
  if (HAL_ADC_Init(&hadc) != HAL_OK)
  {
    Error_Handler();
  }

  /** Configure for the selected ADC regular channel to be converted.
  */
  sConfig.Channel = ADC_CHANNEL_0;
  sConfig.Rank = ADC_RANK_CHANNEL_NUMBER;
  sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
  if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC_Init 2 */

  /* USER CODE END ADC_Init 2 */

}

/**
  * @brief TIM16 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM16_Init(void)
{

  /* USER CODE BEGIN TIM16_Init 0 */

  /* USER CODE END TIM16_Init 0 */

  /* USER CODE BEGIN TIM16_Init 1 */

  /* USER CODE END TIM16_Init 1 */
  htim16.Instance = TIM16;

htim16.Init.Prescaler = 48000-1;
  htim16.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim16.Init.Period = 65536-1;
  htim16.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim16.Init.RepetitionCounter = 0;
  htim16.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim16) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM16_Init 2 */

  /* USER CODE END TIM16_Init 2 */

}

/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART2_Init 0 */

  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 38400;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;

huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pin : LD2_Pin */
  GPIO_InitStruct.Pin = LD2_Pin;

 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
```

```
/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */

/* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
 }
```

```
#endif /* USE_FULL_ASSERT */
```

Program explanation:

According to the configuration in the ioc file, a basic program template is created in STM32Cube IDE which includes initialization of timers, USART, ADC, GPIOS, clock and essential code.

Firstly, a few variables have been declared for data transmission, data reception, and data storage.

Then two user defined functions are created, named adc_output() and end_logging(). adc_output() is used to transmit the output value of ADC and end_logging() clears the values in rx_buffer and stops data from being printed on the terminal.

The main portion of the program starts in while(1) of int_main(). ADC is initiated, and its scaled output is saved in the variable value. The data contained in tx_buffer is constantly transferred and if index 0 in rx_buffer is 'R', then the value at index 1 is checked. Data is logged accordingly if the value received at the first index is ‘a’, ‘b’, ‘c’, or ‘d’; otherwise, data is logged at a default rate (at an interval of 1000ms). When 'S' is received, data logging is halted, and the data in the tx_buffer is once more constantly transferred.

Input:

Analog input is given to the Nucleo board on pins PA0 (ADC input) and GND. The output will be almost double the input due to configuration of the waveform generator.

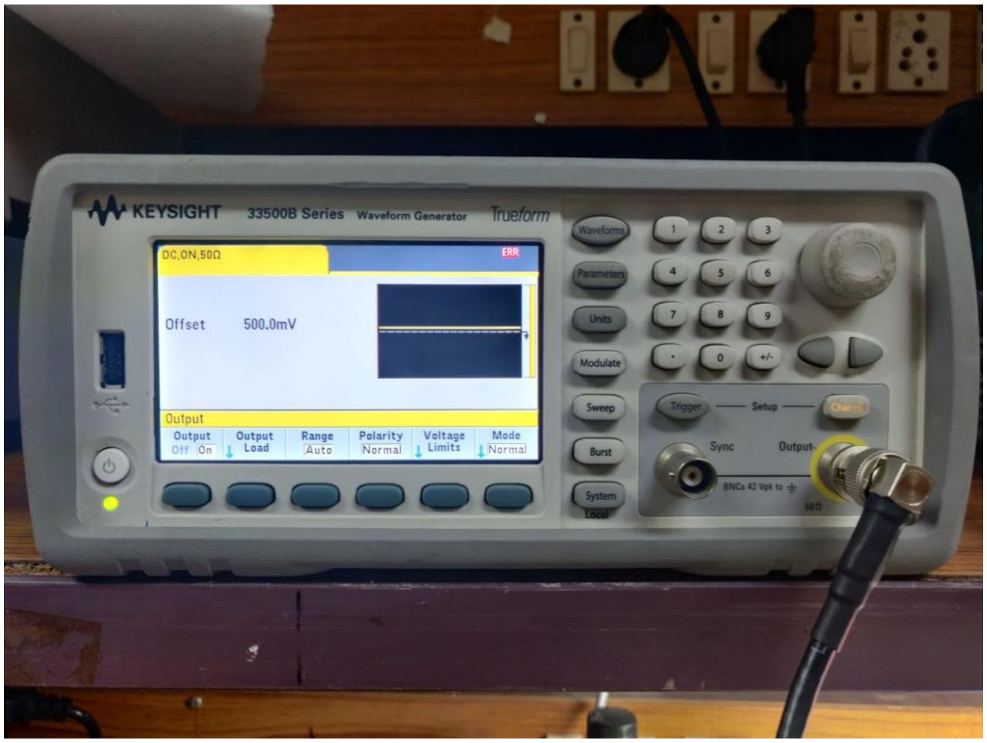


Fig. 2.5 Analog input through waveform generator

Output video link:
https://drive.google.com/file/d/12S3lHEWOeBMwqKYxll19Qj9EMJth3eiV/view?usp=sharing

3. LabView

3.1. Introduction

Graphical Programming Technique is a technique where VISUAL BLOCK Connections are used to code instead of text which makes it easy for non-coders to implement algorithms. LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is the first implementation of graphical programming to date, it continues to be the dominant graphical programming implementation. It provides a powerful and integrated environment for the development of various instrumental applications. An efficient LabVIEW application is designed without unnecessary operations, with minimal occupation including code, data, block diagram, front panel, and GUI updates. It eliminates human errors in data collection and process operations. It reduces data transcription errors and more reliable data available makes better quality control of products and new discoveries. LabVIEW programs are also called virtual instruments (VIs), because their appearance and operation imitate physical instruments.

3.2. Advantages of LabVIEW

Some advantages of this technique over Text-based programming are:

- Graphical programming is highly interactive as compared to Text-based programming.
- In text-based programming, the syntax must be known but in Graphical programming, the syntax is knowledge but not required.
- Front panel design requires extra coding in the case of text-based programming but in the case of Graphical programming, no extra coding is required.
- Errors are indicated as we wire blocks in graphical programming while in Text-based programming, to check error, the program has to be compiled.

3.3. Features of LabVIEW

Some other features of Graphical Programming are:

User-friendly UI: It has a user-friendly drag and drop kind of interactive User Interface.
Built-in Functions: It supports thousands of inbuilt functions that range from analysis and I/O etc. These belong to the function palette.
Scalable: As LabVIEW has a modular design making it easy to scale and modular programs.
Professional Development Tools: It has a plethora of tools that help integrate

and debug large applications.
Open environment: It has tools needed for many open environment developments.
Object–oriented design: It supports object-oriented programming structures enabling encapsulation and inheritance to create modular and extensible code.
Compiled language: Being a compiled language it is faster.

3.4. Graphical User Interface

This is a GUI designed for the program developed in STM32Cube IDE. There are 2 panels in LabView which are displayed below. The block diagram is the panel where the graphical program is generated. The front panel is the GUI which is for the users. According to the program in the block diagram, certain elements appear on the front panel which can be used by the user to operate a system.

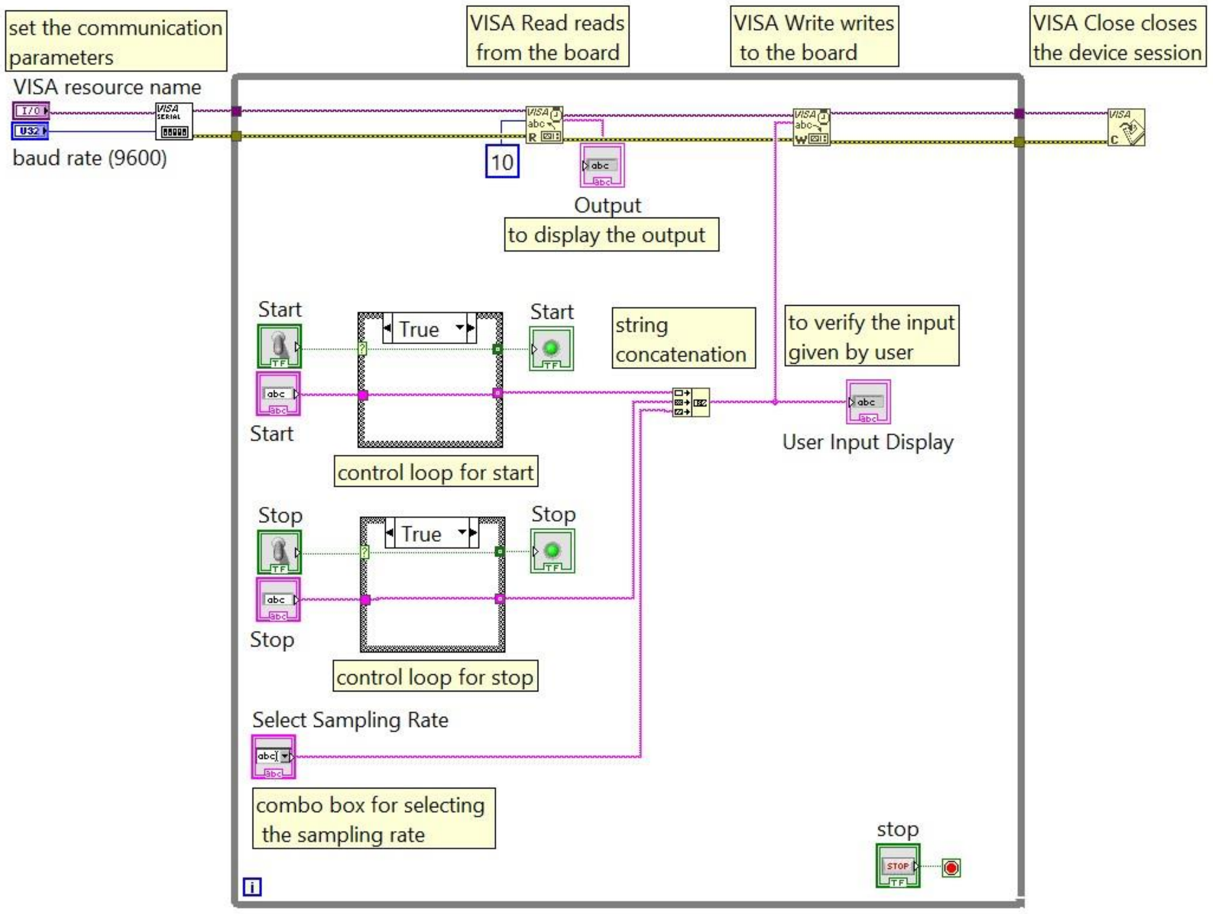


Fig. 3.1 Block Diagram

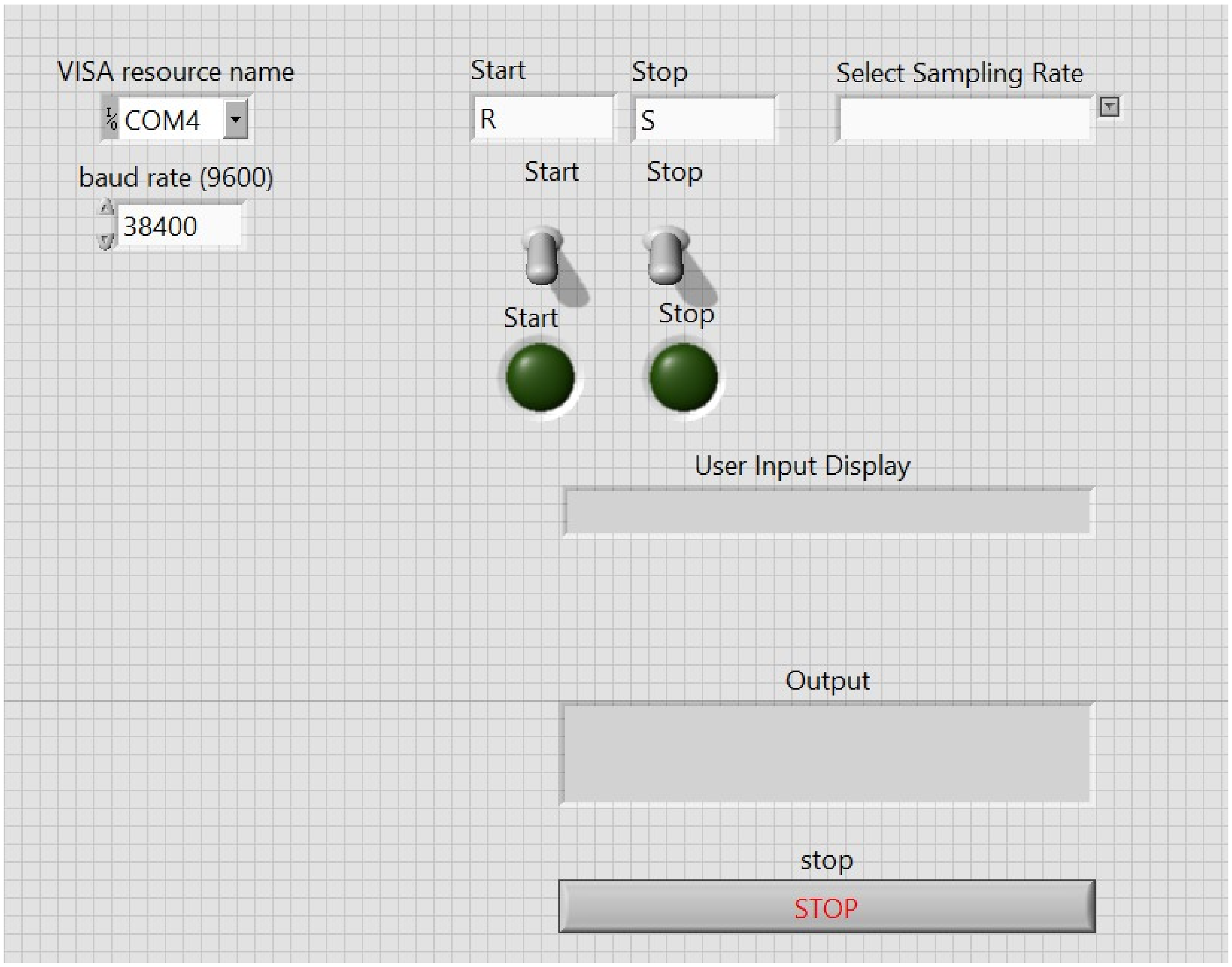


Fig. 3.2 Front Panel

Working:

The communication settings must first be established before changing the COM Port and Baudrate.VISA Read reads the data being transmitted from the board, which, in our case, is the string ‘OK’. On the front panel, the string indicator 'Output' continually displays OK.10 representing the number of bits. After that, there are two boolean buttons, labeled Start and Stop. When the Start button is depressed, the corresponding LED illuminates, 'R' is transmitted to the Nucleo board, and it is also shown in the string indicator 'User Input Display'. When the board receives the signal "R," it begins transmitting the output of the ADC at its default rate, which is shown in "Output". The options for various sampling rates are presented in a dropdown menu using a combo box labeled "Select sampling Rate".Stop functions in a manner similar to Start. The while loop contains the full structure of the code, allowing it to execute constantly until the stop button is pushed. Data transmission and reception are stopped by VISA Close at the end of the while loop structure.

Output video link:
https://drive.google.com/file/d/1495TTapQQuN4lk7kb6aDj1d_o2MpH1n/view?usp=drive_link

4. Applications

Application 1

Develop a data logging system for Nucleo F411RE, which is a development board with ARM Cortex M4 architecture and also develop a GUI for same.

Features:

1. It provides user with a start button

2. An option to vary the sampling rate to 10ms and 1 sec

3. Includes an option to limit the time period of the process to predetermined values

4. It stores the data automatically at the given path into an excel sheet (which is .csv comma delimited or .csv comma separated)

The GUI for the same is created using LabVIEW, as shown in the image below.

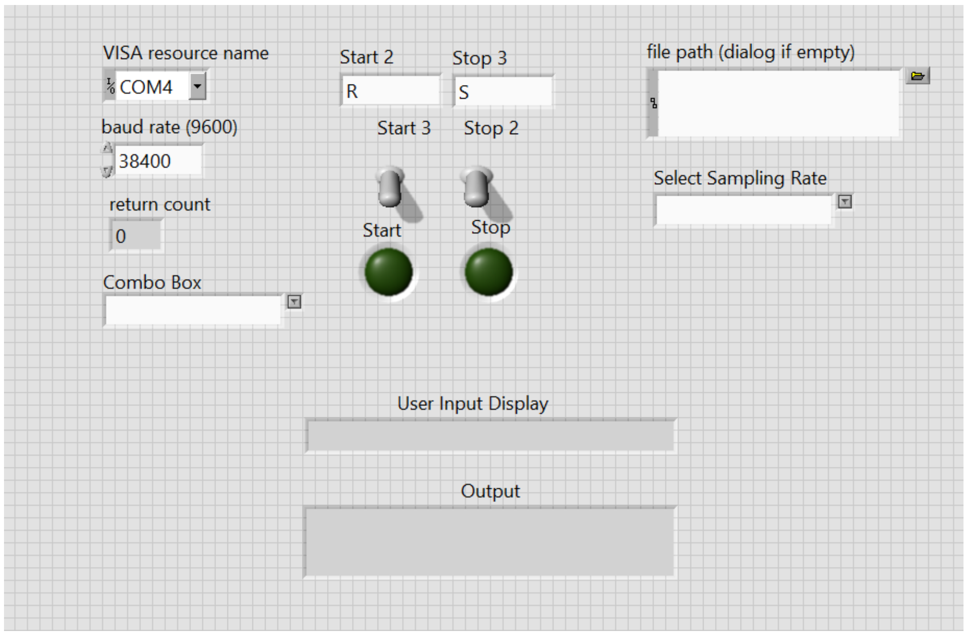


Fig. 4.1

Application 2

Develop a system to log the ADC data at a sampling rate of 200kHz (at every 5 microseconds) to find the slope of the given signal to differentiate a pulse from noise and also to maintain a count of the successful pulses detected.

5. Conclusion

Our project that was an **ARM Controller Based Data Logging System** completed all its objectives of data acquisition and logging. The system developed provides the users the freedom to select the rate at which the ADC output is displayed and determine the time period for which the system remains on.

Overall, the project has demonstrated its effectiveness, reliability, and versatility in capturing and storing data. It has the potential to make a significant impact in various industries and research domains by providing valuable insights and facilitating data-driven decision-making processes.

6. References

- STM32F070RB Datasheet
- https://os.mbed.com/platforms/ST-Nucleo-F070RB/
- https://deepbluembedded.com/stm32-arm-programming-tutorials/